University of Zagreb
Faculty of Electrical Engineering and Computing

Machine Learning 1 2024/2025

http://www.fer.unizg.hr/en/course/maclea1

First lab assignment: Linear Regression. Jaime Rodríguez

Version: 1.0
Last updated: 1. 10. 2021.

Deadline: **20 October 2024, 23:59**

Submission rules

By submitting the exercise, you confirm the following points:

1. You did not receive help from another when solving the exercise;
2. You attributed parts of the code that were taken from the Internet by referencing them in comments;
3. You did not use parts of the code from the Internet that are specific to the laboratory exercise;
4. You have not used UI-assistants for coding such as GitHub Copilot (including generative UI tools such as ChatGPT).

Violation of any of the above rules is considered a misdemeanor and results in academic sanctions.

Instructions

The first lab assignment consists of seven tasks. Follow the instructions in the text cells below. Solving the lab assignment boils down to **supplementing this notebook**: inserting one or more cells **below** the text of the task, writing the appropriate code, and executing the cells.

Make sure you fully understand the code you've written. When submitting the assignment, you must be able to modify and re-execute your code at the request of the teaching assistant. Furthermore, you need to understand the theoretical basis of what you are doing, within the framework of what we covered in the lecture. Below some tasks you can also find questions that serve as guidelines for a better understanding of the material (**do not write** the answers to the questions in the notebook). Therefore, do not limit yourself only to solving the tasks, but feel free to experiment. This is precisely the purpose of these assignments.

You should do the assignment **independently**. You can consult others on the principle way of solving it, but ultimately you have to do the assignment yourself. Otherwise, the assignment makes no sense.

In [5]:
```
# Load the core libraries...

import numpy as np
import sklearn
import matplotlib.pyplot as plt
%pylab inline
```

```
%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib
```

1. Simple regression

You are given a set of labeled examples $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{4} = \{(0,4), (1,1), (2,2), (4,5)\}$. Examples are stored in a matrix $\mathbf{X}$ with dimensions $N \times n$ (in this case 4×1). The corresponding labels are stored in a vector $\mathbf{y}$, with dimensions $N \times 1$ (in this case 4×1):

In [8]:
```
X = np.array([[0],[1],[2],[4]])
y = np.array([4,1,2,5])
```

(a)

Go through the documentation of the function `PolynomialFeatures` from the `sklearn` library. Use that function to generate a design matrix $\mathbf{\Phi}$ in which examples are not mapped into a higher-dimensional feature space (each example is only extended with a *dummy one* feature; $m = n + 1$).

In [10]:
```
from sklearn.preprocessing import PolynomialFeatures
# Your code here
poly = PolynomialFeatures(1)
phy = poly.fit_transform(X)
phy
```

Out[10]:
```
array([[1., 0.],
       [1., 1.],
       [1., 2.],
       [1., 4.]])
```

(b)

Familiarize yourself with the `linalg` module. Calculate the weights $\mathbf{w}$ of a linear regression model as $\mathbf{w} = (\mathbf{\Phi}^\mathsf{T}\mathbf{\Phi})^{-1}\mathbf{\Phi}^\mathsf{T}\mathbf{y}$. Make sure you get the same result if calculating pseudoinverse $\mathbf{\Phi}^+$ of the design matrix, i.e., $\mathbf{w} = \mathbf{\Phi}^+\mathbf{y}$ using the function `pinv`.

In [13]:
```
from numpy import linalg
# Your code here
w = linalg.inv(phy.T @ phy) @ phy.T @ y
print(w)
w = linalg.pinv(phy) @ y
print(w)
```

```
[2.2        0.45714286]
[2.2        0.45714286]
```

For brevity, the vector $\mathbf{x}$ extended with the *dummy one* feature $x_0 = 1$ is denoted as $\tilde{\mathbf{x}}$ in the examples below.

(c)

Calculate the empirical error using the expression $E(h|\mathcal{D}) = \frac{1}{2}\sum_{i=1}^{N}(\tilde{\mathbf{y}}^{(i)} - h(\tilde{\mathbf{x}}^{(i)}))^2$. You can use the mean squared error function `mean_squared_error` from the `sklearn.metrics` module.

Q: Above defined error function $E(h|\mathcal{D})$ and the mean square error function are not completely identical. What is the difference? Which one is more "realistic"?

In [17]:
```
from sklearn.metrics import mean_squared_error
# Your code here
y_pred = phy @ w.reshape((-1,1))
err = mean_squared_error(y_true = y, y_pred = y_pred)
err
```

Out[17]:
```
2.042857142857143
```

(d)

Demonstrate that the weights $\mathbf{w}$ for the examples from $\mathcal{D}$ can not be obtained by solving the system $\mathbf{w} = \mathbf{\Phi}^{-1}\mathbf{y}$, but that we indeed need pseudoinverse.

Q: Why is that the case? Could the problem be solved by mapping the examples into a higher-dimensional feature space? If yes, would that always work, regardless of the set of examples $\mathcal{D}$? Demonstrate on an example.

In [20]:
```
# Your code here

# We can not compute the inverse of phy because is not a square matrix (4 x 2),
# We could try to solve the system of equations by other ways, such as LU factorization
```

(e)

Study the class `LinearRegression` from the `sklearn.linear_model` module. Check if the weights obtained by that function (accessible through `coef_` and `intercept_` attributes) are equal to those you calculated above. If that's not the case, make sure you fix the code so that they are equal.

NB: Pay attention to how the classes `LinearRegression` and `PolynomialFeatures` use intercept and make sure you don't add it multiple times.

Calculate the predictions of the model (`predict` method) and verify that the value of the empirical error is the same as the one you calculated previously.

In [23]:
```
from sklearn.linear_model import LinearRegression
# Your code here
reg = LinearRegression().fit(X,y)
print(f'Parameters calculated with phy: w_0 = {w[0]}, w_1 = {w[1]}')
print(f'Parameters calculated with sklearn: w_0 = {reg.intercept_}, w_1 = {reg.coef_[0]}')
```

```
Parameters calculated with phy: w_0 = 2.199999999999999, w_1 = 0.4571428571428572
Parameters calculated with sklearn: w_0 = 2.2, w_1 = 0.45714285714285713
```

2. Polynomial regression and the noise effect

(a)

Let us now consider the regression on a larger number of examples. Define the function `make_labels(X, f, noise=0)` that takes as input a matrix of unlabeled examples $\mathbf{X}_{N\times n}$ and generates a vector of their labels $\mathbf{y}_{N\times 1}$. Labels are generated as $y^{(i)} = f(x^{(i)}) + \mathcal{N}(0, \sigma^2)$, where $f : \mathbb{R}^n \to \mathbb{R}$ represents the true function that generated the data (in reality, we don't know this function), and σ is the standard deviation of the Gaussian noise, defined with the `noise` parameter. For generating the noise, you can use the function `numpy.random.normal`.

Generate a training set consisting of $N = 50$ examples uniformly distributed in the interval $[-5, 5]$ using the function $f(x) = 5 + x - 2x^2 - 5x^3$ with the noise of $\sigma = 200$:

In [26]:
```
from numpy.random import normal
def make_labels(X, f, noise=0):
    # Your code here
    N = len(X)
    y = np.zeros(N)
    for i in range(N):
        y[i] = f(X[i])[0] + normal(loc= 0.0, scale= noise, size= 1)[0]
    return y
```

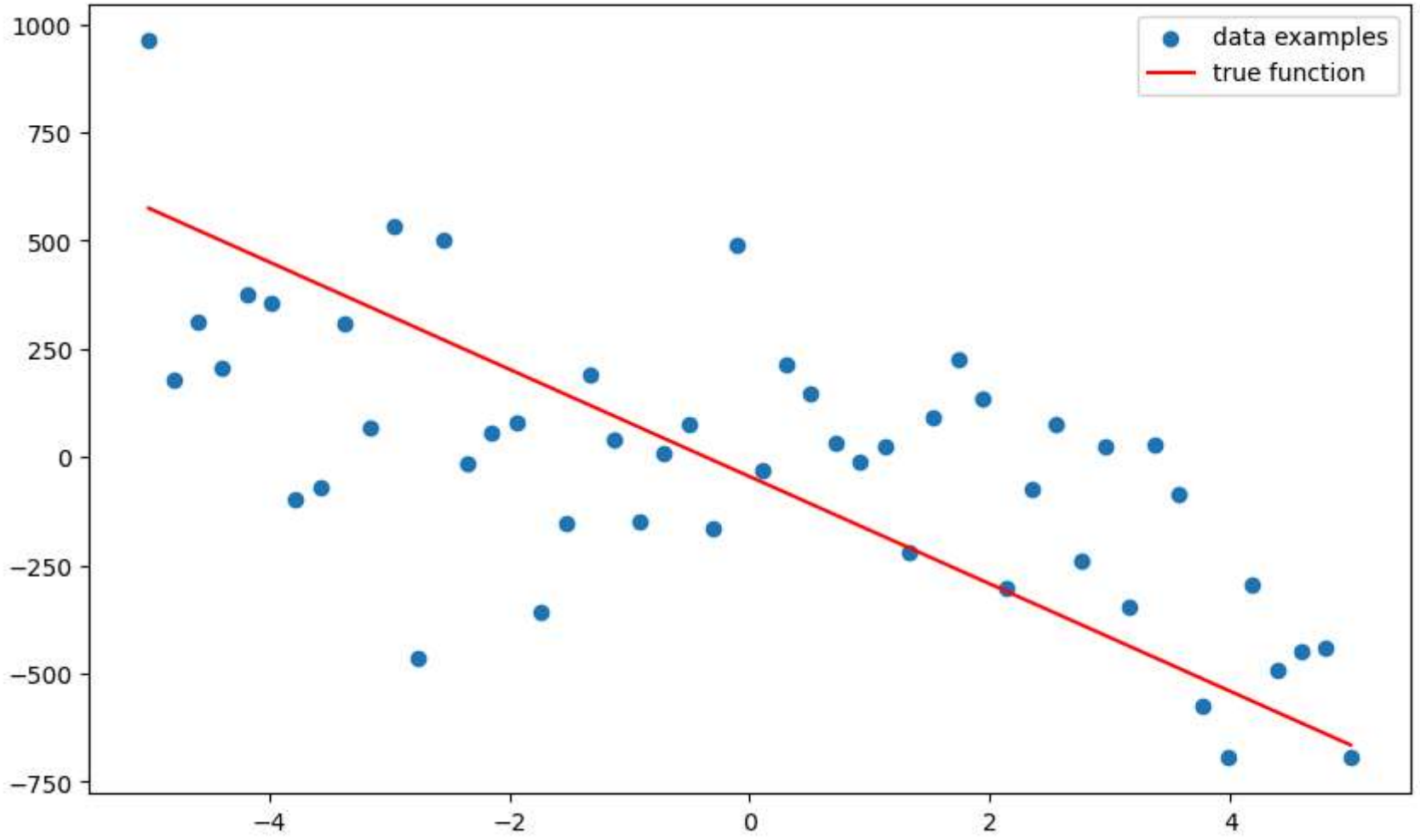
In [27]:
```
def make_instances(x1, x2, N) :
    return np.array([np.array([x]) for x in np.linspace(x1,x2,N)])
```

In [28]:
```
# Your code here
X = make_instances(-5, 5, 50)
```

```
def f(x):
    return 5 + x - 2 * x**2 - 5 * x**3

y = make_labels(X, f, 200)
```

Plot the training set using the `scatter` function.

In [30]:
```
# Your code here
plt.figure(figsize= (10,6))
plt.scatter(X, y)
plt.plot([-5,5], [f(-5),f(5)], c = 'r')
plt.legend(["data examples", "true function"])
plt.show()
```

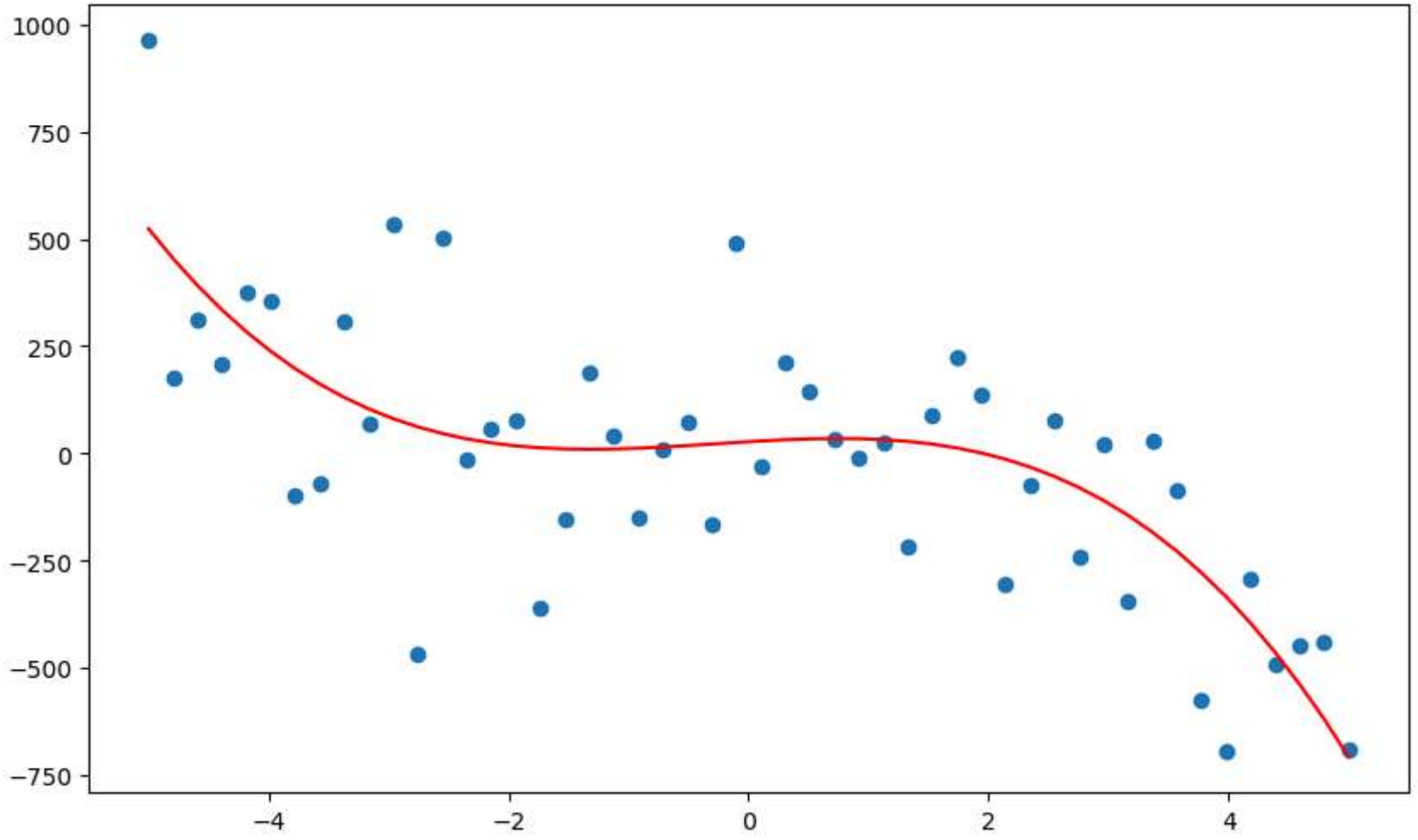


(b)

Train the polynomial regression model of degree $d = 3$. On the same plot, plot the learned model $h(\mathbf{x}) = \mathbf{w}^\intercal \tilde{\mathbf{x}}$ and the training set examples. Calculate the train error of the model.

In [33]:
```
# Your code here
phy = PolynomialFeatures(3).fit_transform(X)
h = LinearRegression().fit(
```

```
    phy,
    y
)

y_pred = h.intercept_ + phy @ h.coef_

plt.figure(figsize= (10,6))
plt.plot(X, y_pred, c= 'r')
plt.scatter(X, y)
plt.show()

err = mean_squared_error(y, y_pred)
print(err)
```




```
47257.664163483714
```

3. Model selection

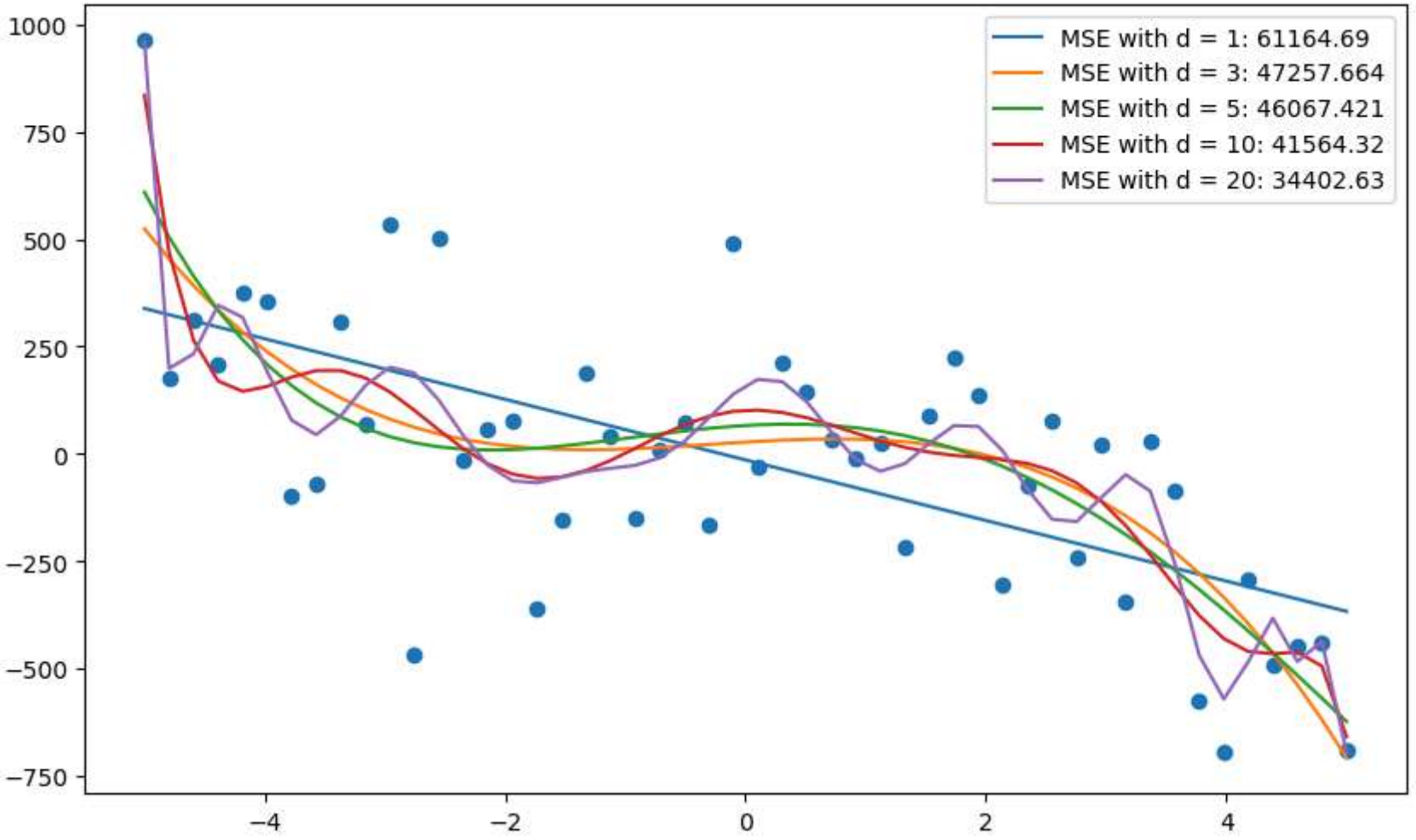
(a)

Using the training set from task 2, train five linear regression models $\mathcal{H}_d$ of varying complexity, where d representes the degree of the polynomial, $d \in \{1, 3, 5, 10, 20\}$. On the same plot, show the training set and the functions $h_d(\mathbf{x})$ for each of the five models (we recommend

that you use `plot` inside the `for` loop). Calculate the train error for each of the models.

Q: Which model has the lowest training error and why?

In [36]:
```
# Your code here
plt.figure(figsize= (10,6))
plt.scatter(X, y)

for d in (1,3,5,10,20):
    phy = PolynomialFeatures(d).fit_transform(X)
    h = LinearRegression().fit(
        phy,
        y
    )
    y_pred = h.intercept_ + phy @ h.coef_
    plt.plot(X, y_pred, label= r'MSE'f' with d = {d}: {round(mean_squared_error(y, y_pred), 3)}')

plt.legend()
plt.show()
```



(b)

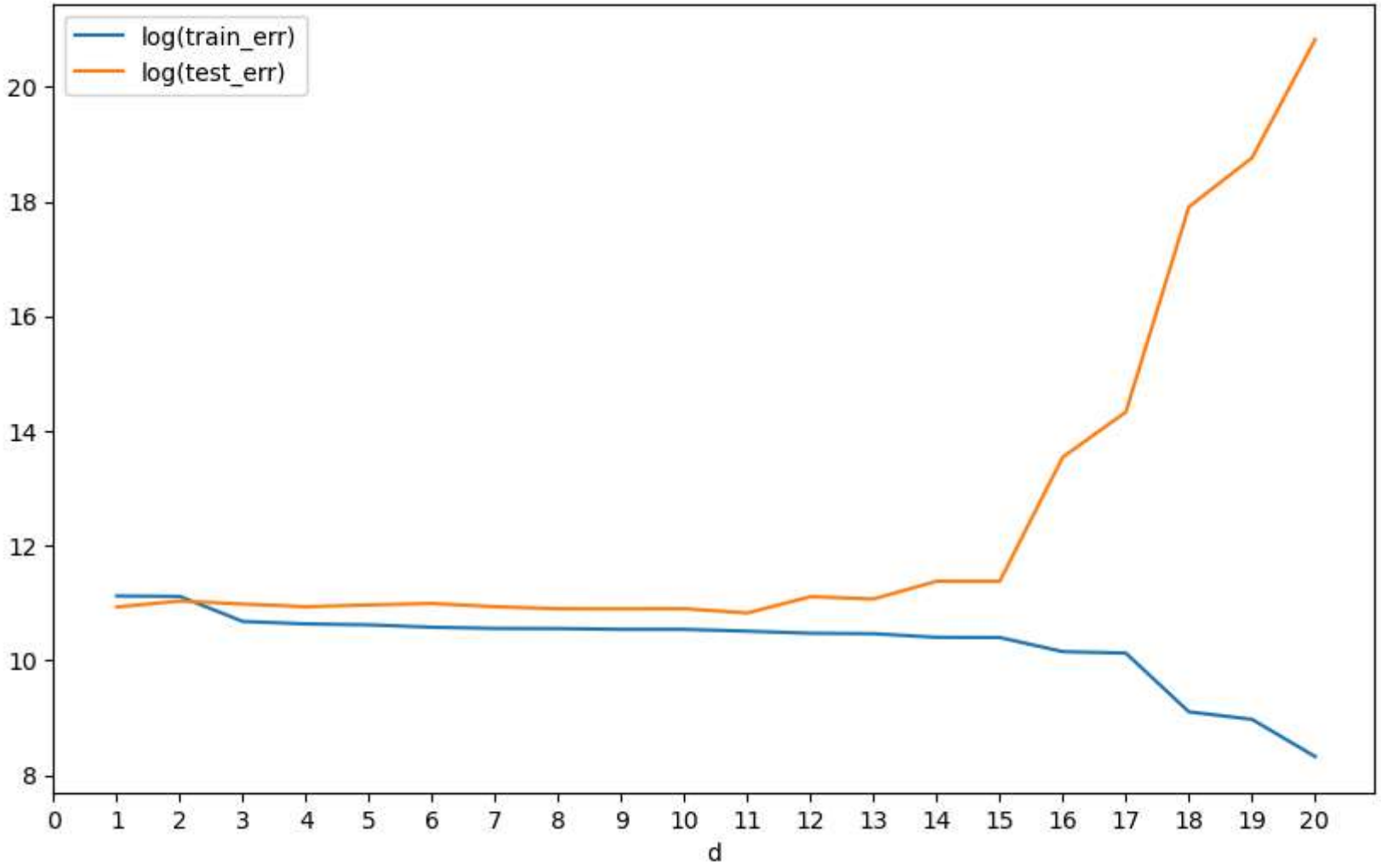
Split the training set from the task 2 into a train and test set in the ratio 1:1 using the function `model_selection.train_test_split`. On the same plot, show the train and the test error for polynomial regression models $\mathcal{H}_d$, with the polynomial degrees in the range $d \in [1, 2, \ldots, 20]$. Since squared error grows fast for larger degrees, instead of plotting the true values of the error function, plot their logarithms.

NB: Train and test splits must be the same for each of the twenty models.

Q: Did the plot turn out as you expected? Which model would you choose among those and why?

Q: Run the cell many times. Do you notice any problem? Would the problem be equally visible if we had more examples? Why?

In [39]:
```
from sklearn.model_selection import train_test_split
# Your code here
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.5)
plt.figure(figsize= (10,6))

train_error = np.zeros(20)
test_error = np.zeros(20)
for d in range(1,21):
    phy_train = PolynomialFeatures(d).fit_transform(X_train)
    phy_test = PolynomialFeatures(d).fit_transform(X_test)
    h = LinearRegression().fit(
        phy_train,
        y_train
    )
    y_pred = h.intercept_ + phy_train @ h.coef_.T
    train_error[d - 1] = mean_squared_error(y_train, y_pred)
    test_error[d - 1] = mean_squared_error(y_test, h.predict(phy_test))

plt.plot(range(1,21), np.log(train_error), label= 'log(train_err)')
plt.plot(range(1,21), np.log(test_error), label= 'log(test_err)')
plt.xlabel('d')
plt.xticks(np.arange(21))
plt.legend()
plt.show()
```



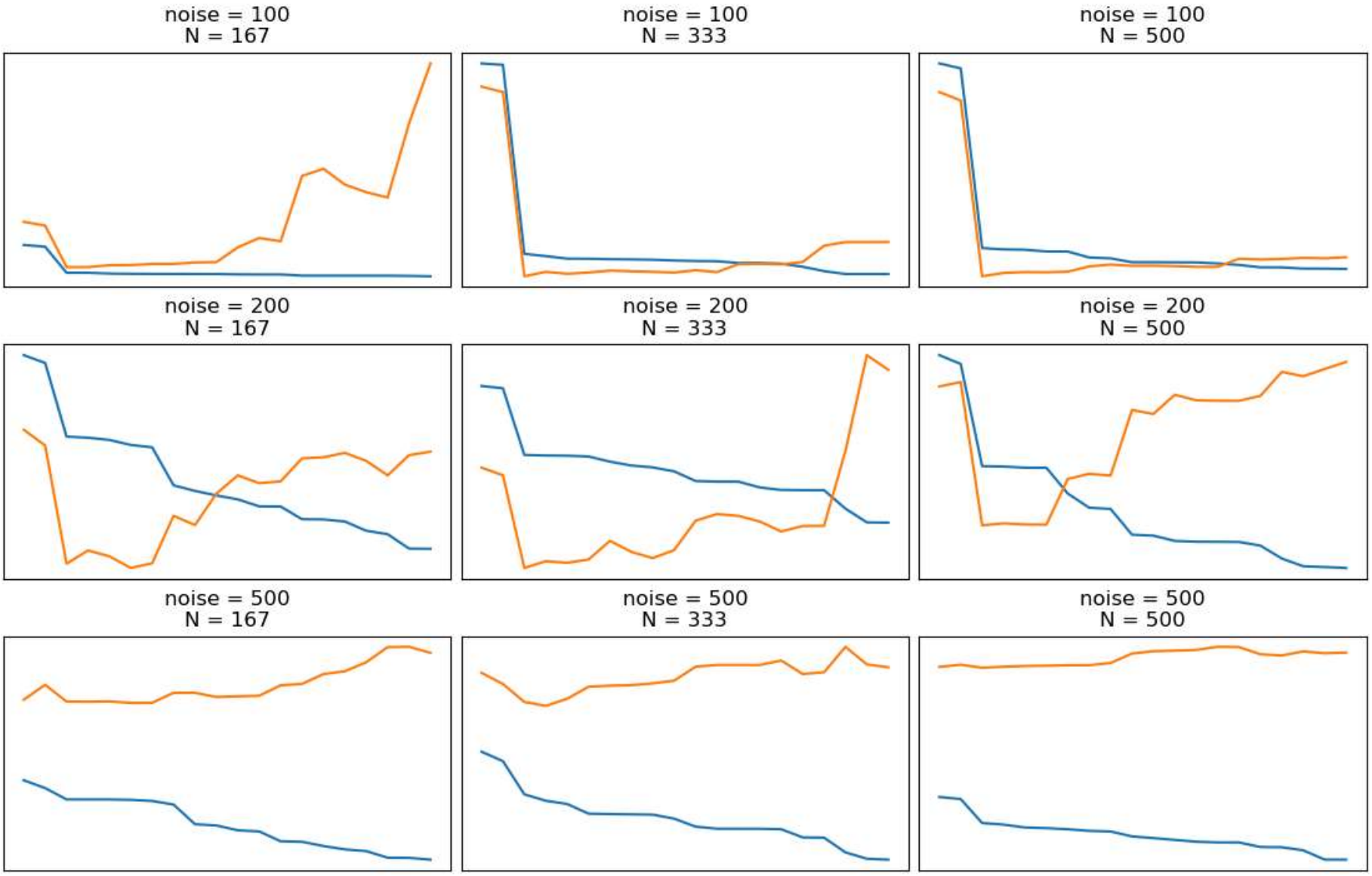
(c)

Accuracy of the model depends on (1) its complexity (degree d of the polynomial), (2) number of examples N, and (3) the amount of noise. In order to analyze these dependencies, plot the errors as in the task 3b, but for different $N \in$ (a third, two thirds, all) and noise amounts $\sigma \in \{100, 200, 500\}$ (9 plots in total). Use the function `subplots` to clearly arrange the plots in a 3×3 table. Data should be generated in the sama way as in the task 2.

NB: Make sure that all the plots are generated on the comparable datasets, in the following manner. First, generate all 1000 examples, split those into train and test sets (two sets, each with 500 examples). Then, create three different versions from both the train and the test set, each version with a different amount of noise (in total 2x3=6 dataset versions). In order to simulate the amount of data (N), using the obtained 6 dataset versions sample a third, two thirds, and all the examples. This leaves you with a total of 18 datasets -- a pair of train and test sets for each of the nine plots.

In [42]:
```
# Your code here
plt.subplots(figsize= (11,7), layout= 'constrained')
plt.axis('off')
current_plot = 1

N = 1000
X = make_instances(-5, 5, N)
X_train, X_test = train_test_split(X, test_size= 0.5)

for noise in (100, 200, 500):
    y_train = make_labels(X_train, f, noise)
    y_test = make_labels(X_test, f, noise)
    for sample_size in (round(1/3 * N / 2), round(2/3 * N / 2), round(N / 2)):
        sample = np.random.choice(N//2, sample_size)
        X_train_sample = X_train[sample] #not necessary in the 3rd loop, just for legibility
        X_test_sample = X_test[sample]
        y_train_sample = y_train[sample]
        y_test_sample = y_test[sample]
        train_error = np.zeros(20)
        test_error = np.zeros(20)
        for d in range(1,21):
            phy_train = PolynomialFeatures(d).fit_transform(X_train_sample)
            phy_test = PolynomialFeatures(d).fit_transform(X_test_sample)
            h = LinearRegression().fit(
                phy_train,
                y_train_sample
            )
            y_pred = h.intercept_ + phy_train @ h.coef_.T
            train_error[d - 1] = mean_squared_error(y_train_sample, y_pred)
            test_error[d - 1] = mean_squared_error(y_test_sample, h.predict(phy_test))
        plt.subplot(3,3,current_plot)
        plt.title(f'noise = {noise}\nN = {sample_size}')
        current_plot += 1
```

```
        plt.plot(range(1,21), np.log(train_error))
        plt.plot(range(1,21), np.log(test_error))
        plt.xticks([])
        plt.yticks([])



plt.show()
```



Q: Did the results turn out as expected? Explain.

4. Regularized regression

(a)

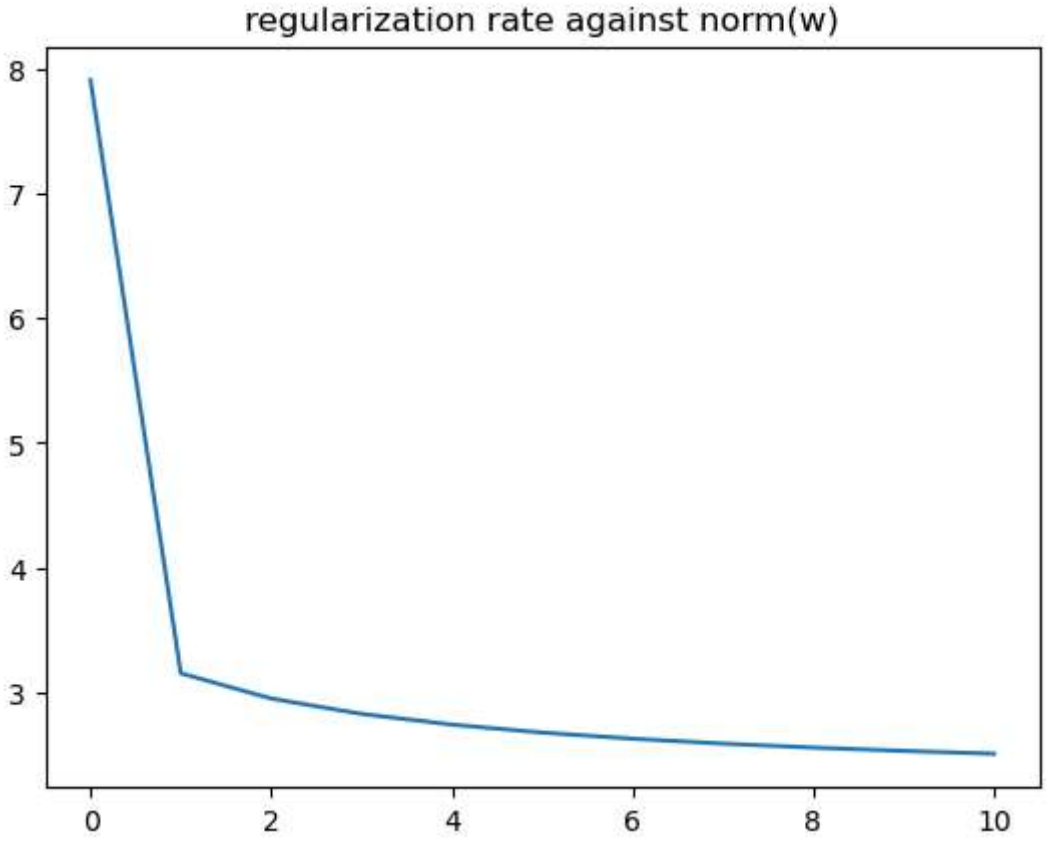
In the experiments above we didn't use **regularization**. Let's go back to the example from the Task 1. Using the examples from that task, calculate the weights $\mathbf{w}$ for the polynomial regression model of the degree $d = 3$ using L2 regularization (known as *ridge regression*), using the expression $\mathbf{w} = (\mathbf{\Phi}^\mathsf{T}\mathbf{\Phi} + \lambda\mathbf{I})^{-1}\mathbf{\Phi}^\mathsf{T}\mathbf{y}$. Calculate the weights using the regularization factors $\lambda = 0$, $\lambda = 1$ and $\lambda = 10$ and compare the obtained weights.

Q: What is the dimension of the matrix that should be inverted?

Q: What are the differences between the obtained weights and are those differences expected? Explain.

In [46]:
```
# Your code here
X = np.array([[0],[1],[2],[4]])
y = np.array([4,1,2,5])
w_norms = np.zeros(11)
phy = PolynomialFeatures(3).fit_transform(X)
for reg_rate in range(0,11):
    reg_term = np.eye(phy.shape[0])
    reg_term[0,0] = 0 # we dont regularize intercept term
    reg_term *= reg_rate
    w = linalg.inv(phy.T @ phy + reg_term) @ phy.T @ y
    w_norms[reg_rate] = np.linalg.norm(w)
    if(reg_rate in [0,1,10]):
        print(w)

plt.plot(np.arange(11), w_norms)
plt.title('regularization rate against norm(w)')
plt.show()
```

```
[ 4.         -5.91666667  3.375      -0.45833333]
[ 3.05696145 -0.69079365 -0.2831746   0.1445805 ]
[ 2.49444184 -0.15897295 -0.13423067  0.0815601 ]
```



(b)

Study the `Ridge` class from the `sklearn.linear_model` module, which implements L2-regularized regression model. Parameter α corresponds to the λ parameter. Apply this model on the same examples as in the previous task and print out the obtained weights $\mathbf{w}$ (attributes `coef_` and `intercept_`). Again, pay attention to the intercept.

Q: Are the weights identical to those from the Task 4a? If not, explain why is that and how would you fix it?

In [49]:
```
from sklearn.linear_model import Ridge
# Your code here
for reg_rate in (0, 1, 10):
    clf = Ridge(alpha = reg_rate)
```

```
    clf.fit(phy,y)
    print(clf.intercept_)
    print(clf.coef_)
```

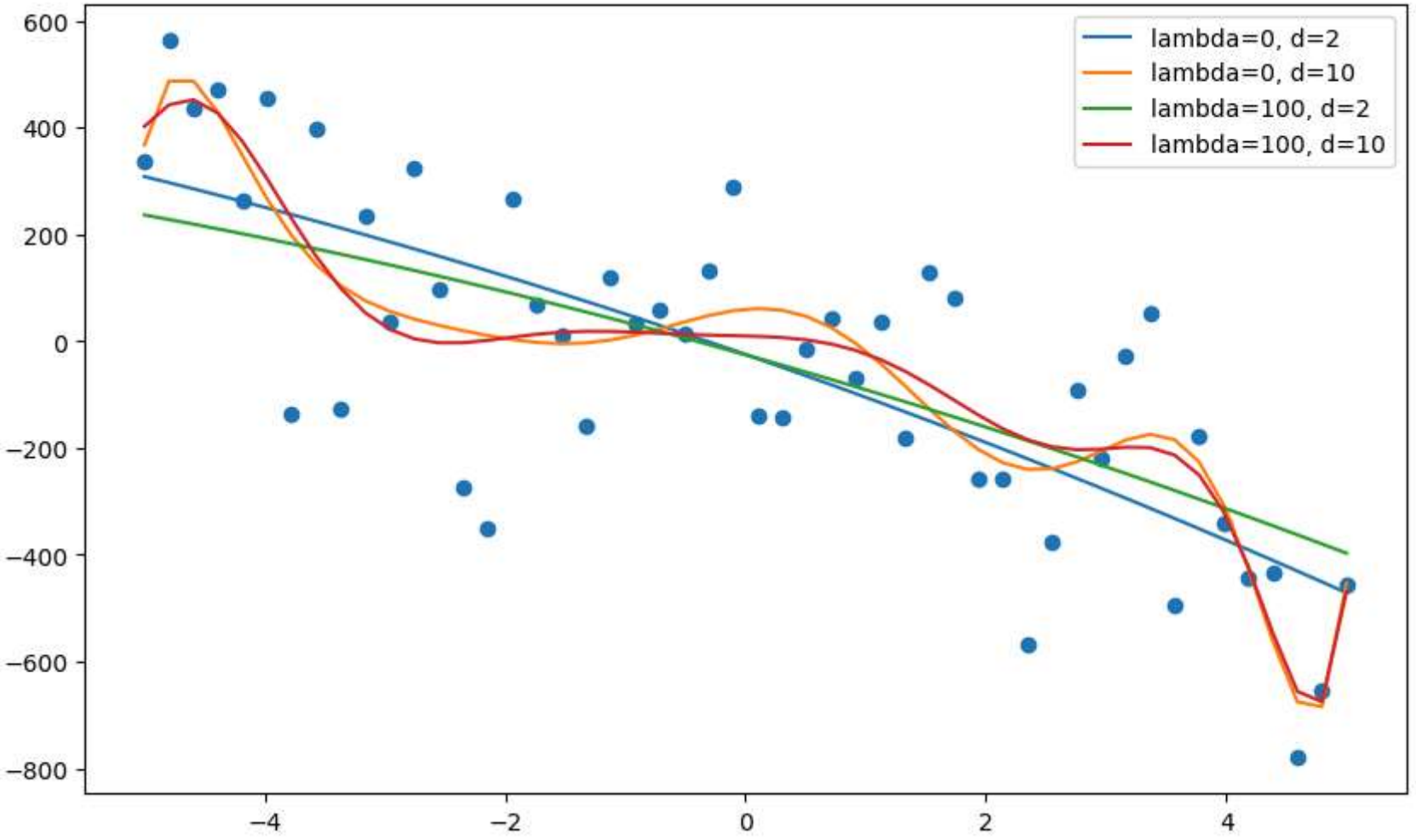
```
4.000000000000025
[ 0.         -5.91666667  3.375      -0.45833333]
3.0569614512471652
[ 0.         -0.69079365 -0.2831746   0.1445805 ]
2.4944418431229733
[ 0.         -0.15897295 -0.13423067  0.0815601 ]
```

(c)

Let's go back to the case of $N = 50$ randomly generated examples from the Task 2. Train the polynomial regression models $\mathcal{H}_{\lambda,d}$ for $\lambda \in \{0, 100\}$ and $d \in \{2, 10\}$ (four models in total). Plot the corresponding functions $h(\mathbf{x})$ and the examples (on the same plot; we recommend that you use `plot` inside a `for` loop).

Q: Are the results that you obtained expected? Explain.

In [51]:
```
# Your code here
X = make_instances(-5, 5, 50)
y = make_labels(X, f, 200)

plt.figure(figsize= (10,6))
plt.scatter(X, y)

for reg_rate in [0,100]:
    for d in [2,10]:
        phy = PolynomialFeatures(d).fit_transform(X)
        h = Ridge(alpha= reg_rate).fit(phy, y)
        y_pred = h.intercept_ + phy @ h.coef_
        plt.plot(X, y_pred, label= f'lambda={reg_rate}, d={d}')

plt.legend()
plt.show()
```



(d)

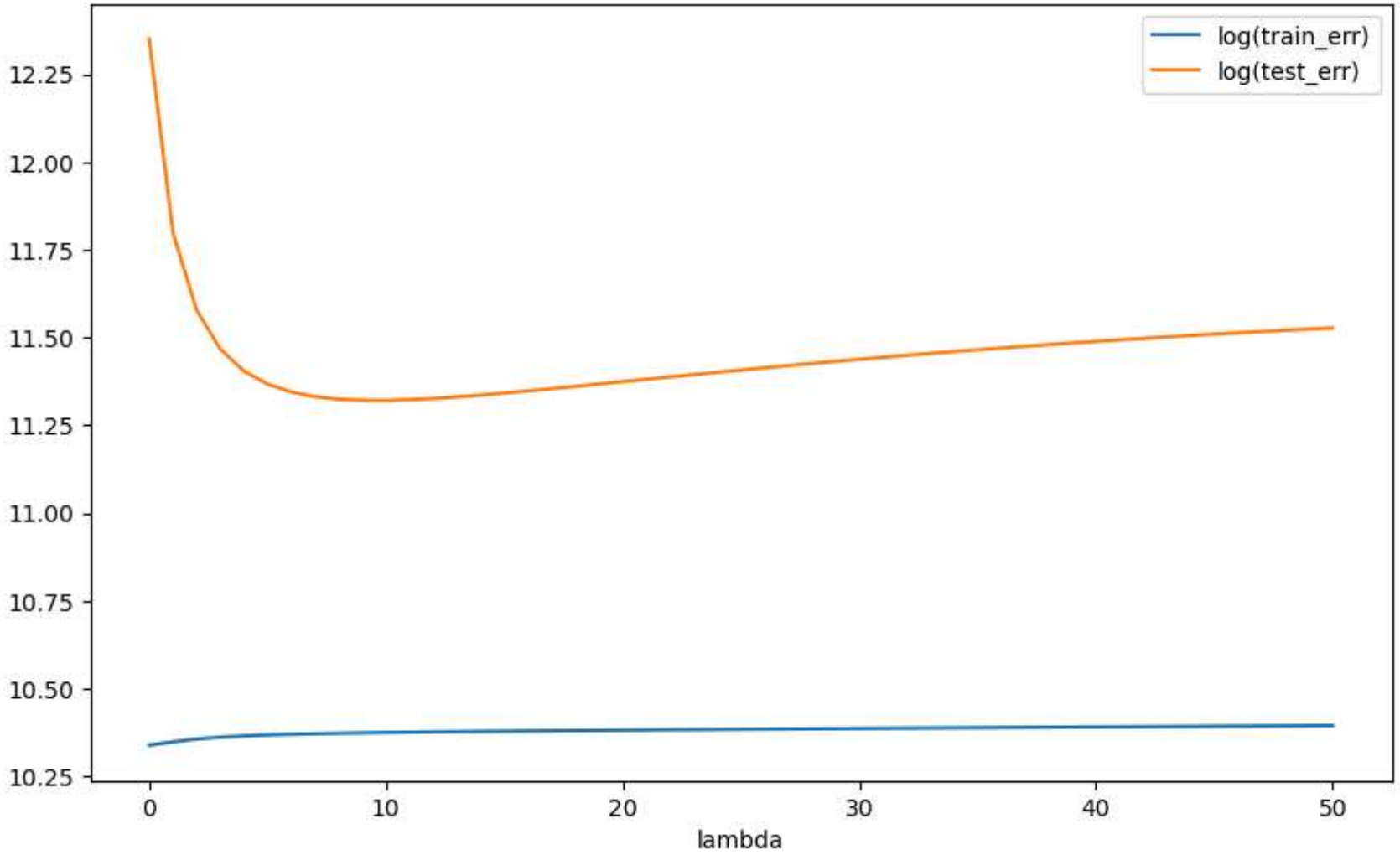
As in the Task 3b, split the dataset into a train and a test set in the ratio 1:1. Plot the logarithms of the train and test errors with respect to the model $\mathcal{H}_{d=10,\lambda}$, varying the regularization factor λ in the range $\lambda \in \{0, 1, \ldots, 50\}$.

Q: Which area of the plot corresponds to the overfitting and which one to the underfitting? Why?

Q: Which value for λ would you choose based on these plots and why?

In [53]:

```
# Your code here
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.5)
plt.figure(figsize= (10,6))

train_error = np.zeros(51)
test_error = np.zeros(51)
d = 10
for reg_rate in range(0,51):
    phy_train = PolynomialFeatures(d).fit_transform(X_train)
    phy_test = PolynomialFeatures(d).fit_transform(X_test)
    h = Ridge(alpha= reg_rate).fit(phy_train, y_train)
    y_pred = h.intercept_ + phy_train @ h.coef_.T
    train_error[reg_rate] = mean_squared_error(y_train, y_pred)
    test_error[reg_rate] = mean_squared_error(y_test, h.predict(phy_test))



plt.plot(range(0,51), np.log(train_error), label= 'log(train_err)')
plt.plot(range(0,51), np.log(test_error), label= 'log(test_err)')
plt.legend()
plt.xlabel('lambda')
plt.show()
```



5. L1-regularization and L2-regularization

The goal of regularization is to push down to zero as many of the model's weights $\mathbf{w}$ as possible in order to obtain a simpler model. Model's complexity can be characterized using the norm of the model's weights vector $\mathbf{w}$, typically using L2 or L1 norm. For a trained model we can also determine the number of non-zero features, or L0 norm, using the following function that takes a weights vector $\mathbf{w}$ as input:

In [56]:
```
def nonzeroes(coef, tol=1e-6):
    return len(coef) - len(coef[np.isclose(0, coef, atol=tol)])
```

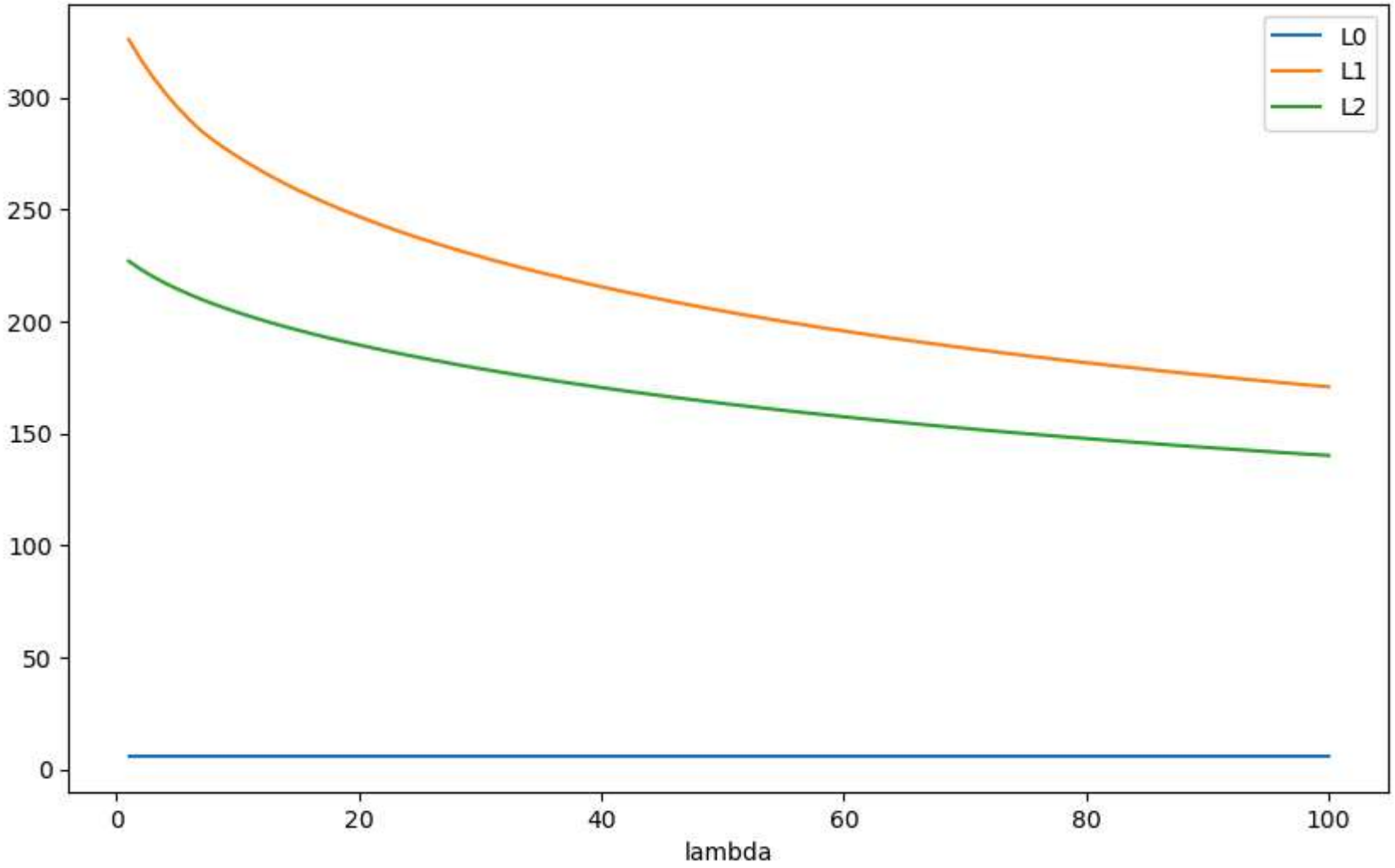
(a)

For this task, use the train and test sets from the Task 3b. Train **L2-regularized** polynomial regression models of degree $d = 5$, varying the hyperparameter λ in the range $\{1, 2, \ldots, 100\}$. For each of the trained models, calculate the L{0,1,2} norms of the weight vector $\mathbf{w}$. Plot the values of the norms as a function of λ. Pay attention to what exactly is passed to the function for the norm calculation.

Q: Explain the shape of the obtained curves. Will the curve for $||\mathbf{w}||_2$ ever reach zero? Why? Is that a problem? Why?

Q: For $\lambda = 100$, what percentage of the weights is equal to zero, or, in other words, how sparse is the model?

In [58]:
```
from sklearn.linear_model import Ridge
# Your code here
X = make_instances(-5, 5, 50)
y = make_labels(X, f, 200)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.5)
plt.figure(figsize= (10,6))
norms = np.zeros((3, 100))
d = 5
for reg_rate in range(1,101):
    phy_train = PolynomialFeatures(d).fit_transform(X_train)
    phy_test = PolynomialFeatures(d).fit_transform(X_test)
    h = Ridge(alpha= reg_rate).fit(phy_train,y_train)
    w = h.coef_
    w[0] = h.intercept_
    norms[0, reg_rate - 1] = nonzeroes(w)
    norms[1, reg_rate - 1] = np.linalg.norm(w, 1)
    norms[2, reg_rate - 1] = np.linalg.norm(w, 2)

plt.plot(range(1,101), norms[0,:], label= 'L0')
plt.plot(range(1,101), norms[1,:], label= 'L1')
plt.plot(range(1,101), norms[2,:], label= 'L2')
plt.xlabel('lambda')
plt.legend()
plt.show()
```

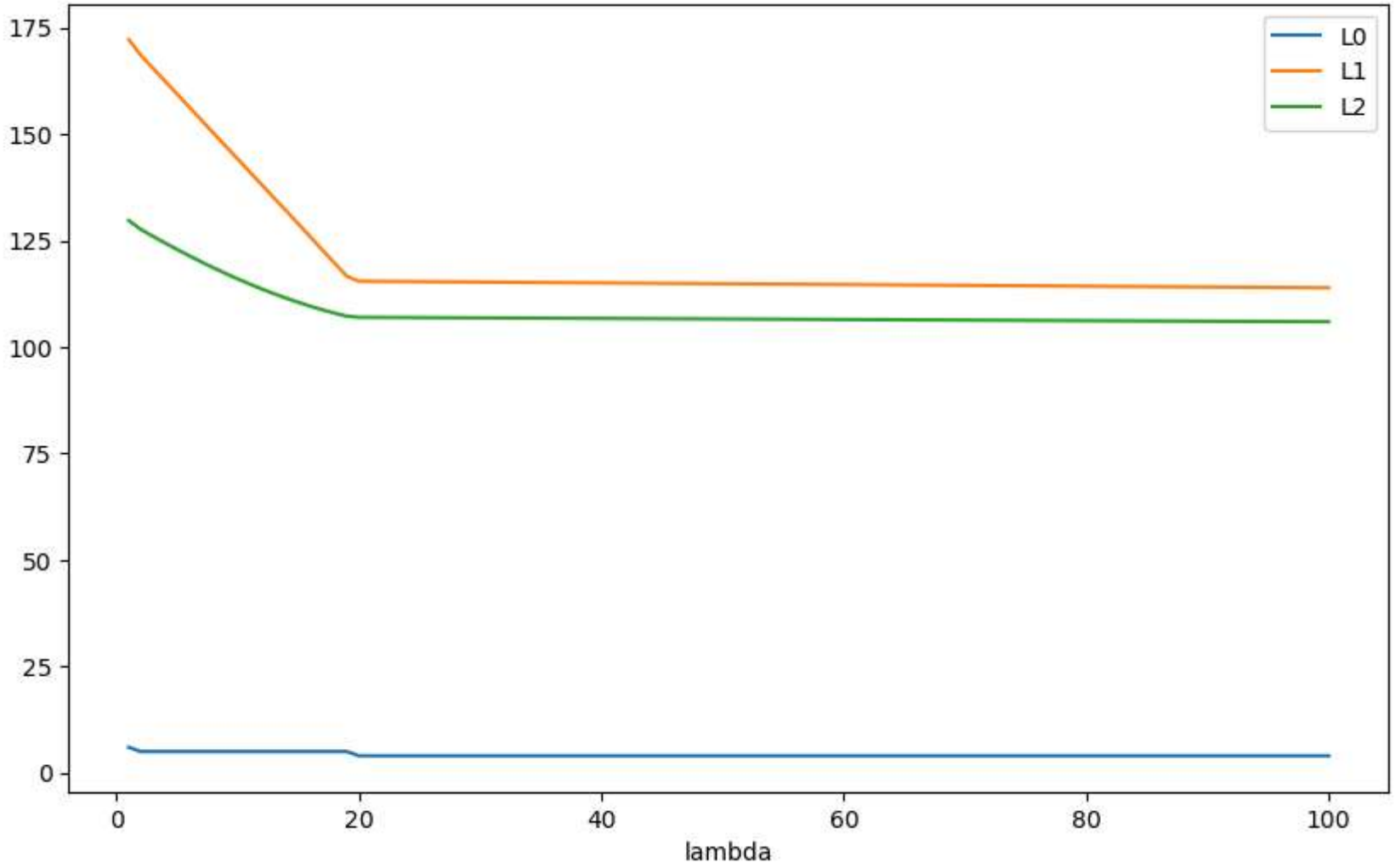


(b)

The main advantage of the L1 regularized regression (or *LASSO regression*) over the L2 regularized regression is that L1 regularized regression results in **sparse models**, i.e., the models that have many weights pushed to zero. Show that this really happens by running the experiment from above using **L1 regularized** regression, implemented in the class `Lasso` in the package `sklearn.linear_model`.

In [61]:

```
from sklearn.linear_model import Lasso
# Your code here
X = make_instances(-5, 5, 50)
y = make_labels(X, f, 200)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.5)
plt.figure(figsize= (10,6))
norms = np.zeros((3, 100))
d = 5
for reg_rate in range(1,101):
    phy_train = PolynomialFeatures(d).fit_transform(X_train)
    phy_test = PolynomialFeatures(d).fit_transform(X_test)
    h = Lasso(alpha= reg_rate).fit(phy_train,y_train)
    w = h.coef_
    w[0] = h.intercept_
    norms[0, reg_rate - 1] = nonzeroes(w)
    norms[1, reg_rate - 1] = np.linalg.norm(w, 1)
    norms[2, reg_rate - 1] = np.linalg.norm(w, 2)

plt.plot(range(1,101), norms[0,:], label= 'L0')
plt.plot(range(1,101), norms[1,:], label= 'L1')
plt.plot(range(1,101), norms[2,:], label= 'L2')
plt.xlabel('lambda')
plt.legend()
plt.show()
```



6. Features of different scales

In practice, we often encounter the datasets where not all features are of the same magnitude. An example of such dataset is the regression dataset `grades` in which the task is to predict student's average grade at college (1--5) based on two features: number of points that student obtained on the entry exam (1--3000) and student's average grade in high school. Average grade on college is calculated as a weighted sum of the two features with added noise.

Use the following code to generate such dataset.

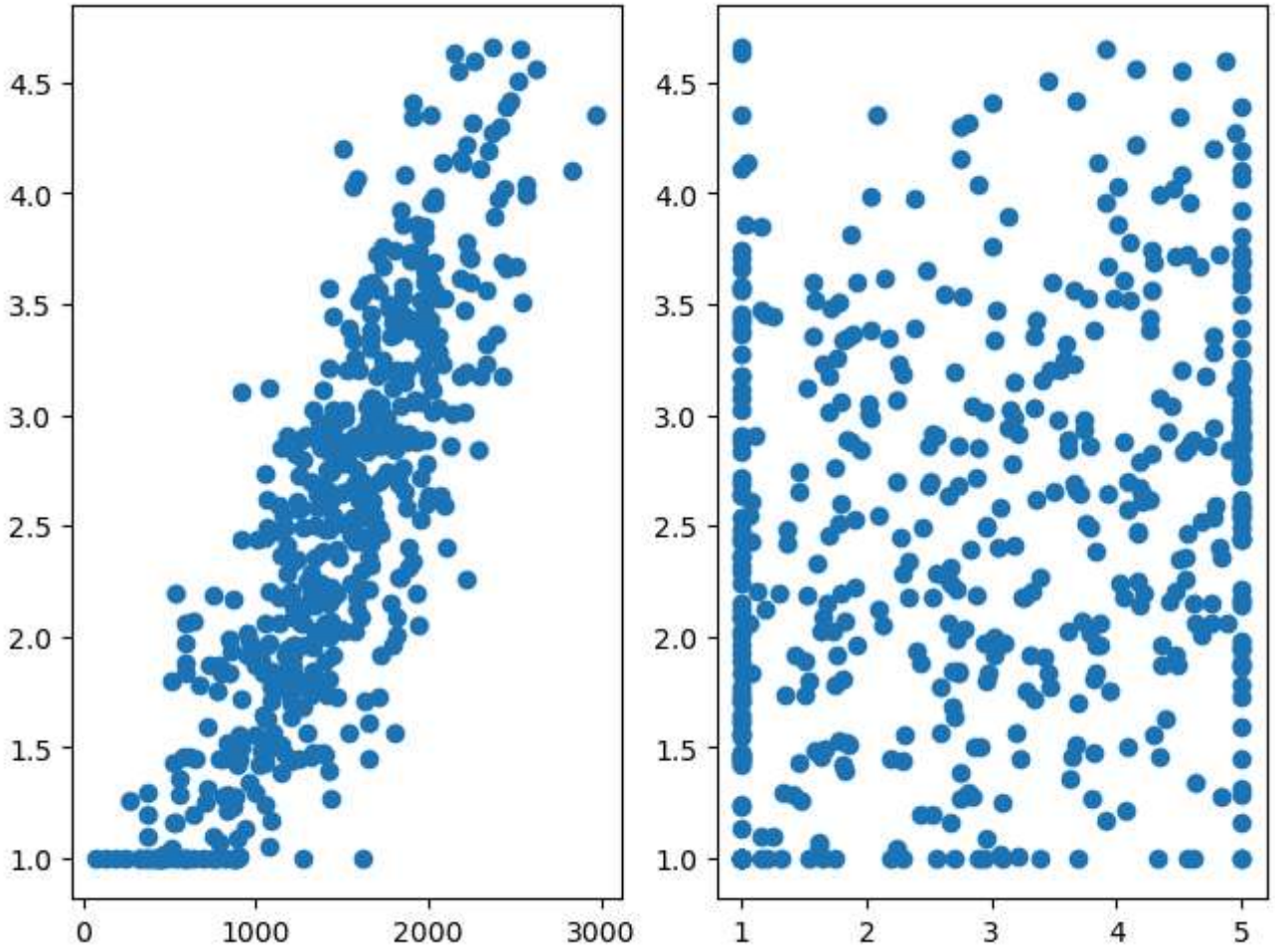
In [64]:
```
n_data_points = 500
np.random.seed(69)

# Generate the data of entry exam points using normal distribution and limit the points to the interval [1, 3000].
exam_score = np.random.normal(loc=1500.0, scale = 500.0, size = n_data_points)
exam_score = np.round(exam_score)
exam_score[exam_score > 3000] = 3000
exam_score[exam_score < 0] = 0

# Generate the data of high school grades using normal distribution and limit them to the interval [1, 5].
grade_in_highschool = np.random.normal(loc=3, scale = 2.0, size = n_data_points)
grade_in_highschool[grade_in_highschool > 5] = 5
grade_in_highschool[grade_in_highschool < 1] = 1

# Design matrix.
grades_X = np.array([exam_score,grade_in_highschool]).T

# Finally, generate the output values (average college grade).
rand_noise = np.random.normal(loc=0.0, scale = 0.5, size = n_data_points)
exam_influence = 0.9
grades_y = ((exam_score / 3000.0) * (exam_influence) + (grade_in_highschool / 5.0) \
            * (1.0 - exam_influence)) * 5.0 + rand_noise
grades_y[grades_y < 1] = 1
grades_y[grades_y > 5] = 5
```

a)

Plot the relationship between the dependent variable (y-axis) and the first feature, as well as the second feature (x-axis). Plot two separate graphs.

In [67]:
```
# Your code here
plt.subplots(layout= 'constrained')
plt.axis('off')

plt.subplot(121)
plt.scatter(exam_score, grades_y)
```

```
plt.subplot(122)
plt.scatter(grade_in_highschool, grades_y)

plt.show()
```



b)

Train the L2 regularized regression model ($\lambda = 0.01$) using the data in `grades_X` and `grades_y`:

In [70]:
```
# Your code here
model = Ridge(alpha= 0.01)
model.fit(grades_X, grades_y)
print(model.coef_)
```

```
[0.00141497 0.09477276]
```

Now repeat the experiment from the above, but first scale the data `grades_X` and `grades_y` and store those into variables `grades_X_fixed` and `grades_y_fixed`. For this, use `StandardScaler`.

In [72]:
```
from sklearn.preprocessing import StandardScaler
# Your code here
scaler = StandardScaler()
scaler.fit(grades_X)
grades_X_fixed = scaler.transform(grades_X)

scaler = StandardScaler()
scaler.fit(grades_y.reshape(-1,1))
grades_y_fixed = scaler.transform(grades_y.reshape(-1,1))

model = Ridge(alpha= 0.01)
model.fit(grades_X_fixed, grades_y_fixed)
print(model.coef_)
```

```
[[0.81630375 0.15167761]]
```

Q: Looking at the plots from the subtask (a), which feature should have a larger magniture, or, in other words, higher importance when predicting college average grade? Do the obtained weights support your intuiton? Explain.

7. Multicollinearity

a)

Duplicate the last column (average high school grade) in the dataset `grades_X_fixed` and store this new dataset into variable `grades_X_fixed_colinear`. This effectively introduces perfect multicollinearity.

In [77]:
```
# Your code here
grades_X_fixed_colinear = np.zeros((grades_X_fixed.shape[0], 3))
grades_X_fixed_colinear[:,0] = grades_X_fixed[:,0]
grades_X_fixed_colinear[:,1] = grades_X_fixed[:,1]
grades_X_fixed_colinear[:,2] = grades_X_fixed[:,1]
print(grades_X_fixed_colinear[:5, :])
```

```
[[ 0.95063817 -0.78607869 -0.78607869]
 [-0.50343434 -0.50193004 -0.50193004]
 [ 1.18596832 -0.52213172 -0.52213172]
 [-0.50152109  1.00664465  1.00664465]
 [-1.45432123  1.07657545  1.07657545]]
```

Again, train the L2 regularized regression model ($\lambda = 0.01$), but now using the dataset `grades_X_fixed_colinear`.

In [79]:
```
# Your code here
model = Ridge(alpha= 0.01)
model.fit(grades_X_fixed_colinear, grades_y_fixed)
print(model.coef_)
```

```
[[0.81630364 0.07583957 0.07583957]]
```

Q: Compare the weights with those obtained in the Task *7b*. What is the difference?

b)

Randomly sample 50% of the examples from the dataset `grades_X_fixed_colinear` and train two L2 regularized regression models, one with $\lambda = 0.01$ and the other with $\lambda = 1000$. Repeat this experiment 10 times (each time with a different subsample of 50% of the examples). For each model, print out the obtained weights vector in each of the 10 repetitions. Also, print out the standard deviation for each weight (six standard deviations in total, each obtained over 10 values).

In [83]:
```
# Your code here
N = grades_X_fixed_colinear.shape[0]
W = np.zeros((10,6))

for i in range(10):
    sample = np.random.choice(N, N//2)
    print(f'iteration {i+1}')
```

```
    model = Ridge(alpha= 0.01)
    model.fit(grades_X_fixed_colinear[sample,:], grades_y_fixed[sample])
    W[i,:3] = model.coef_
    W[i,0] = model.intercept_[0]
    print(f'w for lambda=0.01: {W[i,:3]}')

    model = Ridge(alpha= 1000)
    model.fit(grades_X_fixed_colinear[sample,:], grades_y_fixed[sample])
    W[i,3:] = model.coef_
    W[i,3] = model.intercept_[0]
    print(f'w for lambda=1000: {W[i,3:]}\n\n')

print('standar deviations:')
print(np.std(W, axis=0))
```

```
iteration 1
w for lambda=0.01: [-0.04694136  0.05798842  0.05798842]
w for lambda=1000: [-0.10386646  0.05277113  0.05277113]


iteration 2
w for lambda=0.01: [0.01321924 0.10853086 0.10853086]
w for lambda=1000: [-0.04533824  0.03682682  0.03682682]


iteration 3
w for lambda=0.01: [-0.04806159  0.09258294  0.09258294]
w for lambda=1000: [-0.027286    0.02992725  0.02992725]


iteration 4
w for lambda=0.01: [0.02188085 0.0516362  0.0516362 ]
w for lambda=1000: [0.04965628 0.02508137 0.02508137]


iteration 5
w for lambda=0.01: [-0.02457653  0.07886073  0.07886073]
w for lambda=1000: [0.00557495 0.03428091 0.03428091]


iteration 6
w for lambda=0.01: [-0.05610365  0.06556442  0.06556442]
w for lambda=1000: [-0.06057746  0.02379852  0.02379852]


iteration 7
w for lambda=0.01: [0.00373011 0.06199831 0.06199831]
w for lambda=1000: [-0.09544347  0.02447929  0.02447929]


iteration 8
w for lambda=0.01: [-0.00480907  0.07953089  0.07953089]
w for lambda=1000: [-0.0011519   0.04243913  0.04243913]


iteration 9
```

```
w for lambda=0.01: [-0.01621528  0.08006777  0.08006777]
w for lambda=1000: [-0.07384675  0.03982161  0.03982161]


iteration 10
w for lambda=0.01: [0.03976242 0.06472009 0.06472009]
w for lambda=1000: [0.05750082 0.01845126 0.01845126]


standar deviations:
[0.0306667  0.01639009 0.01639009 0.05379203 0.00991688 0.00991688]
```

Q: How regularization affects the weights stability?

Q: Are the coefficients of the same magnitude as in the previous experiment? Explain why.